Computer Architecture - CS2323. Autumn 2024

Lab-4 (RISC-V Simulator)

Report

In the main function, we have used hash maps to create a mapping between the opcode and the type of instructions. Mainly three types of mappings have been developed:

- **M_opcode:** relation between the command name and its corresponding 7-bit opcode.
- **M_format:** relation between the7-bit opcode and its corresponding format type (as a char).
- **M_funct3:** relation between the command name and its corresponding 3-bit funct3.

Additionally, we have used a mapping to relate the register aliases for easy access to their standard format register values from x0-x31.

Utility functions used:

- **Trim():** to remove extra spaces before and after the line
- **toBinary():** to convert the input no to a specific length binary string
- **getFirstWord():** to get the first substring before a delimiter
- **reset** function is made which resets the following:
    - PC
    - Registers
    - Break points
    - Memory
    - Instructions

We have created separate functions to handle instruction strings for different formats of instructions. The basic parameters in each of these functions are m_format, m_opcode, m_funct3, alias (the mapping of alias registers), line, command and the line number.

Without loss of generality, we have used a while loop to extract the relevant information into the variables. At every iteration, we are trimming the line using the trim() function and updating the starting point of the line using the ‘find’ function to obtain the index of the delimiters expected for that format.

Labels: The labels is being identified in the (label_lineno) map data structure, wherein for every label its corresponding line number is being stored, which is used in B and J format instructions.

For e. g.

beq x4, x7, L1

sh x5, -2048(x6)

L1: xor a5, a3, a7

Additionally, we have created:

- A vector of long long which stores all the register values.
- A set of long long which stores all the break points given by the user. A set has been used so that we don't store the same break points again, and it is easier to check whether the line number is a break point or not, and similarly when we need to del a break point from "del break" functionality, we can easily erase that line from the set.
- A reset function is made which resets the following:
    - PC
    - Registers
    - Break points
    - Memory
    - Instructions
- We create a memory using vector of bitset wherein each bitset is of size 8 bytes, and the overall size of the memory is 0x50002.

Execution of Various Functions:

- **Load Command:** The load command loads all the instructions from the given input file in the vector of strings instructions, at every load command, it resets all the previous instructions before loading.
- **Run Command:** It performs every instruction line by line unless it encounters a break point or the instructions are over, it updates the PC and the registers at every instruction, depending upon the instruction format, the value of registers are updated and the PC is updated accordingly.
- **Regs Command**: It prints all the value of registers at once.
- **Exit Command:** It gracefully exits the simulator.
- **Step Command:** Runs only one instruction at which the PC is currently present and updates the PC accordingly.
- **Mem Command:** Starting from the given start PC, it prints all the memory locations for the given number of lines.
- **Break Command:** Sets a mark to stop the code execution once the line is reached, preserving registers and memory state, we use a set for storing all the break points, while running any instructions, we check whether it is present in the set or not.
- **Del Break Command:** we check whether the given breakpoint is actually present in the set or not initially, if yes then we delete that breakpoint from the set.

Execution of commands for various Instructions:

- **R Instruction:** Once we get the different components of the instruction, we use the mappings to get the register variables in standard format. For preventing wrong input registers, we check if the obtained value is present in the alias mapping containing all the acceptable values. The instruction set for this format contains commands {add, sub, and, or, xor, sll, srl, sra }. Depending on the command, we change the value at the destination register and update the PC.

- **I-format:** We have two broad categories in this instruction format, We use two types of functions for handling the inputs of these 2 cases and processing them. We also check the correctness of the immediate value entered. We also check if for the shifing operations, the no of bits being shifted is within bounds.
    - Without Memory: The instruction format without memory are: addi, xori, ori, andi, slli, srli, srai, Depending on the command, we change the value at the destination register and update the PC..

    - With Memory: The instructions with memory are: lb,lh,lw, ld, lbu, lhu, ldu, jalr, int these instructions, based on the number of bytes to accessed, while following the little Endian notation, we find the value for each byte and add it to the answer shifted by 8 bits iteratively, for unsigned type of instructions, we are making the bytes not to be included in the answer as zero.

      Note: For jalr instructions, we are following the format **jalr rd, rs1(n)**

- **U-format**: In the while loop for analysing input, we get the values of operation, destination register and immediate value. The instruction set for this format contains command lui. In case of lui, the RIPES simulator gives error on negative values and values greater than – 1. The PC is incremented by 4 to point to the next instruction. The computed 32-bit immediate value is stored in the destination register (rd). Thus, reg[rd] is updated with the sign-extended immediate value, and pc is incremented by 4 for normal instruction flow.

- **S-format:** A store_type function is made which stores a value into memory starting at a address, byte by byte, for the specified number of bytes. It breaks the value into 8-bit chunks and shifts it right to process the next byte. The store_handle function calculates the effective memory address using the value in register rs1 and an offset, then stores the value from register rs2 into memory based on the store

instruction (sb, sh, sw, or sd), which determines how many bytes are written (1, 2, 4, or 8).

- **B-format**: The B_format function handles branch instructions (beq, bne, blt, bge, bltu, bgeu) by comparing the values of registers rs1 and rs2. For {beq, bne, blt, and bge} it compares the signed values of val1 and val2, updating the PC to the target address if the condition is met. For {bltu and bgeu} we store them in unsigned long long values, and then comparing them based upon their sign, and updates PC based on the result. If none of the conditions are satisfied, the PC is incremented by 4 to proceed to the next instruction.

- **J-format:** It stores the current PC + 4 (the address of the next instruction) into register rd. Then, the PC is updated to the address corresponding to the target label, allowing the program to jump to that location.

Testing:

The code written was tested against various possible cases at each point for each kind of instruction format to detect errors. The output produced by our code was checked against the output of the RIPES simulator.